Time-Parameterized Queries in Spatio-Temporal Databases

Yufei Tao
Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
http://www.cs.ust.hk/~taoyf/

Dimitris Papadias
Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
http://www.cs.ust.hk/~dimitris/

ABSTRACT

Time-parameterized queries (TP queries for short) retrieve (i) the *actual result* at the time that the query is issued, (ii) the *validity period* of the result given the current motion of the query and the database objects, and (iii) the *change* that causes the expiration of the result. Due to the highly dynamic nature of several spatio-temporal applications, TP queries are important both as standalone methods, as well as building blocks of more complex operations. However, little work has been done towards their efficient processing. In this paper, we propose a general framework that covers time-parameterized variations of the most common spatial queries, namely window queries, *k*-nearest neighbors and spatial joins. In particular, each of these TP queries is reduced to nearest neighbor search where the distance functions are defined according to the query type. This reduction allows the application and extension of well-known branch and bound techniques to the current problem. The proposed methods can be applied with mobile queries, mobile objects or both, given a suitable indexing method. Our experimental evaluation is based on R-trees and their extensions for dynamic objects.

Keywords

Spatio-temporal databases, nearest neighbor queries

1. INTRODUCTION

As opposed to traditional, "instantaneous", queries that are evaluated only once to return a single result, *continuous* queries may require constant evaluation and update of the results as the query conditions or database contents change [TGNO92, CDTW00]. Such queries are especially relevant to spatio-temporal databases, which are inherently dynamic and the result of any query is strongly related to the temporal context. An example of a continuous spatio-temporal query is: "based on my current direction and speed of travel, which will be my nearest two gas stations for the next 5 minutes?". A result of the form <{A,B},[0,1)>, <{B,C},[1,5)> would imply that A,B will be the two nearest neighbors during interval [0,1), and B, C afterwards. Notice that the corresponding instantaneous query ("which are my nearest gas stations now?") is usually meaningless in highly dynamic environments; if the query point or the database objects move, the result may be invalidated immediately.

ACM SIGMOD'2002, June 4-6, Madison, Wisconsin, USA.

Any spatial query has a continuous counterpart whose termination clause depends on the user or application needs. Consider, for instance, a window query, where the window (and possibly the database objects) moves/changes with time. The termination clause may be temporal (for the next 5 minutes), a condition on the result (e.g., until exactly one object appears in the query window, or until the result changes three times), a condition on the query window (until the window reaches a certain point in space) etc. A major difference from continuous queries in the context of traditional databases, is that in case of spatio-temporal databases, the object's dynamic behavior does not necessarily require updates, but can be stored as a function of time using appropriate indexes [BJSS98, TUW98, KGT99, AAE00, SJLL00]. Furthermore, even if the objects are static, the results may change due to the dynamic nature of the query itself (i.e., moving query window), which can be also represented as a function of time. Thus, a spatio-temporal continuous query can be evaluated instantly (i.e., at the current time) using time-parameterized information about the dynamic behavior of the query and the database objects, in order to produce several results, each covering a validity period in the future.

The building block of most continuous spatio-temporal queries is what we call the *time-parameterized (TP) query*. A TP query returns: (i) the objects that satisfy the corresponding spatial query, (ii) the *expiry time* of the result, and (iii) the *change* that causes the expiration of the result. As an example, consider that a moving user wants to find all hotels within a 5km range from his/her current position. In addition to a set of hotels (lets say A,B,C) currently within the 5km range, the result contains the time (e.g., 1 minute) that this answer set is valid (given the direction and the speed of the user's movement), as well as the new answer set after the change (e.g., at 1 minute hotel D will start to be within 5km). In the previous example we assume that the query window is dynamic and the database objects are static. In other cases the opposite may be true, e.g., find all cars that are within a 5km range from hotel A. It is also possible that both the query and the objects are dynamic, if for instance, the query and the database objects are points denoting moving airplanes. The same concept can be applied to other common query types, e.g., nearest neighbors and spatial joins (find all major residential areas currently covered by typhoons, together with the earliest time that the situation is expected to change).

TP queries, as standalone methods, are crucial in applications involving dynamic environments (e.g., location-based commerce for mobile communications, air-traffic control systems), where any result should be accompanied by an expiry period in order to be effective in practice. In addition, they constitute the primitive components based on which complex continuous queries can be constructed. In this paper we propose a general framework for TP queries in spatio-temporal databases, which can be applied for any

query type, and any query/object mobility combination (i.e., dynamic queries, dynamic objects, or both). In particular, we show that all time-parameterized queries can be reduced to some form of nearest neighbor search and processed accordingly. The various query types are differentiated by the definitions of distance functions used in each case. As a second step we extend our techniques to solve general continuous and other queries.

The rest of the paper is organized as follows: section 2 surveys related work, while section 3 discusses TP variations of spatial queries and their transformations to nearest neighbor search. Section 4 extends our approach to continuous and "earliest event" queries. Section 5 presents an extensive experimental evaluation, while section 6 concludes with directions for future work.

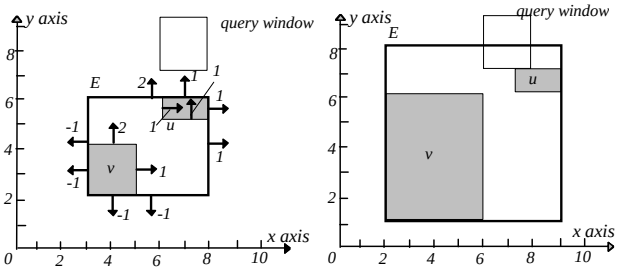
2. RELATED WORK

Despite the importance of continuous queries in spatio-temporal databases, and the bulk of research that has been carried out on traditional queries (e.g., nearest neighbors, spatial joins), there is limited work on the efficient processing of spatio-temporal continuous queries. In [SWCD97], the authors focus on modeling and query languages but do not propose access or processing methods. Song and Roussopoulos [SR01] process moving nearest neighbor queries in R-trees by employing sampling. That is, they incrementally compute the results at pre-determined positions, using previous results to avoid total re-computation. This approach is limited in scope (only applicable to nearest neighbors, and static objects). Furthermore, it suffers from the usual drawbacks of sampling, i.e., if the sampling rate is low the results will be incorrect, otherwise there is a significant computational overhead; in any case there is no accuracy guarantee since even a high sampling rate may miss some results. Zheng and Lee [ZL01] discuss an even more restricted version of the problem (moving query, static objects indexed by R-trees) for a single nearest neighbor, using Voronoi diagrams. In addition to the NN of the query point, they return the valid period of the result, which is a conservative approximation obtained by assuming that the query can have a maximum speed. Neither approach can deal with dynamic objects or other types of queries.

The proposed techniques significantly extend previous work, both in terms of effectiveness and applicability to far more general problems. Although our methods can be employed with any data-partition structure, we consider that the underlying indexes are based on R-tree variants, due to their popularity. In particular static objects are indexed by R*-trees [BKSS90], and dynamic objects by TPR-trees [SJLL00]. Assuming that the reader is familiar with R*-trees, in section 2.1 we describe the TPR-tree. Section 2.2 outlines branch and bound algorithms, which constitute the core of our query processing techniques.

2.1 The Time Parameterized R-tree (TPR-tree)

The TPR-tree [SJLL00] is an extension of the R-tree that can answer prediction queries on dynamic objects. A dynamic object is represented with (i) a minimum bounding rectangle (MBR) that bounds its extents at the current time, and (ii) a velocity vector. Figure 2.1a shows the representation of two objects *u* and *v*, and that of the node that contains them. The arrows indicate the velocity directions for each edge, while the numbers correspond to their values. Velocities towards the negative direction of a coordinate axis are negative. Notice that different edge velocities will cause an object to grow or shrink with time (object *v*). Similarly, an intermediate entry also stores an MBR and its velocity vector. As in traditional R-trees, the extents are such that the MBR tightly encloses all entries in the node at the current time (see entry *E* in Figure 2.1a).



(a) Boundaries at current time 0 (b) Boundaries at future time 1

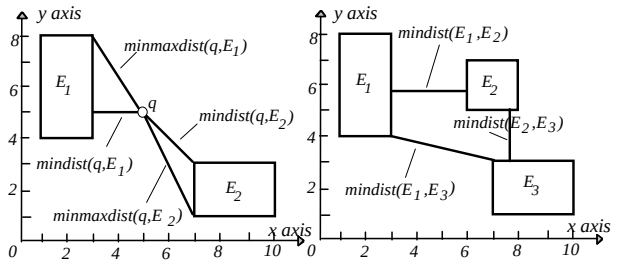
Figure 2.1: Representation of entries in the TPR-tree

The velocity vector of the (intermediate) MBR is determined as follows: (i) the velocity of the upper (right) edge is the maximum of all velocities on this dimension in the sub-tree; (ii) the velocity of the lower (left) edge is the minimum of all velocities on this dimension. This ensures that the MBR always encloses the underlying objects, but it is not necessarily tight. Figure 2.1b shows *u*, *v* and enclosing node *E* at time 1 (observe how the extents and positions of *u*, *v*, *E* change). Since the upper edge of *E* moves with speed 2 (the speed of the upper edge of *v*) the MBR of *E* is not tight. Future MBRs (e.g., in Figure 2.1b) are not stored explicitly, but are computed based on the current extents and velocity vectors.

The TPR-tree answers instantaneous queries at some future time, e.g., retrieve the objects that will intersect the query window at time 1. Such queries are processed in exactly the same way as in the R-tree, except that the extents of the MBRs at the query time are first calculated dynamically and then compared with the query window. Node *E* must be visited because its computed MBR (and entry *u*) intersects the query, although its MBR at the current time does not.

2.2 Branch-and-bound (BaB) Algorithms

The first R-tree BaB algorithm was proposed in [RKV95] for nearest neighbor (NN) queries. The algorithm introduces two distance metrics (both defined on intermediate entries) for pruning the search space. The first metric, *mindist*, is the minimum distance between the query object *q* and any object that can be in the subtree of entry *E*. The second metric, *minmaxdist*, refers to the minimum distance from *q* within which an object in the subtree of *E* is guaranteed to be found. Figure 2.2a illustrates these two metrics on the MBRs of E_1 and E_2 with respect to a point query *q*.



(a) *mindist* & *minmaxdist* (b) *mindist* of two rectangles

Figure 2.2: Pruning metrics

The algorithm of [RKV95] answers a NN query by traversing the R-tree in a depth-first (DF) manner. Specifically, starting from the root, all entries are sorted according to their *mindist* from the query point, and the entry with the lowest value is visited first. The process is repeated recursively until the leaf level where the first potential nearest neighbor is found. During backtracking to the upper levels, the algorithm only visits entries whose *mindist* is smaller than the distance of the nearest neighbor already found. As an example consider the R-tree of Figure 2.3, where the number in each entry refers to the *mindist* (for intermediate entries) or the actual distance (for point objects) from the query point (these numbers are not stored but computed dynamically during query processing). DF would first visit the node of root entry E_1 (since it has the minimum *mindist*), and then the node of E_4, where the first candidate object (*a*) is retrieved. When backtracking to the previous level, entry E_6 is excluded since its *mindist* is greater than the distance of *a*, but E_5 has to be visited before backtracking again at the root level. *Minmaxdist* (and other similar bounds) can be applied to further prune search.

The performance of DF was shown to be suboptimal in [PM97], which reveals that an optimal NN search algorithm only needs to visit those nodes whose MBRs intersect the so-called "search region", i.e., a circle centered at the query point with radius equal to the distance between the query and its nearest neighbor (shaded circle in Figure 2.3). Based on this, [CPZ98, WSB98, BBK+01] investigate cost models for performing NN queries in high-dimensional space.

A best-first (BF) algorithm for KNN query processing using R-trees is proposed in [HS99]. BF keeps a *heap* with the entries of the nodes visited so far. Initially the heap contains the entries of the root sorted according to their *mindist*. In Figure 2.3 when E_1 is visited, it is removed from the heap and the entries of its node (E_4, E_5, E_6) are added together with their *mindist*. The next entry visited is E_2 (it has the minimum *mindist* in the heap), followed by E_8, where the actual result (*h*) is found and the algorithm terminates. BF is optimal in the sense that it only visits the nodes necessary for obtaining the nearest neighbor. Its performance in practice, however, may suffer from buffer thrashing if the available memory is not enough for the required heap. In this case part of the heap must be migrated to the disk, which may incur frequent disk accesses.

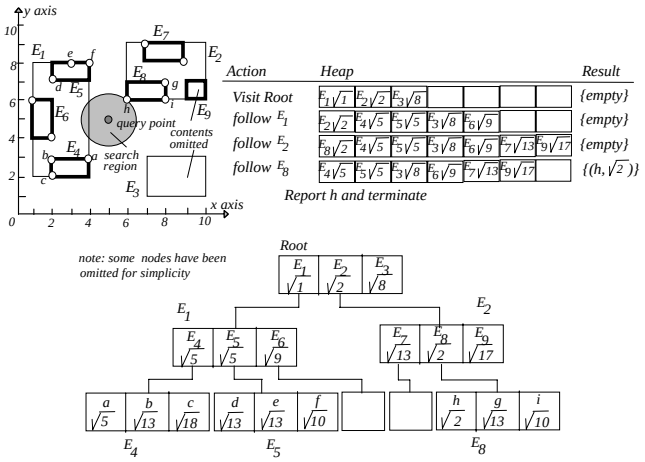


Figure 2.3: Example of BaB algorithms

The BaB framework also applies to closest pair queries that find the pair of objects from two datasets, such that their distance is the minimum among all pairs. Corral et al, [CMTV00] propose various algorithms based on the concepts of DF and BF traversal. The difference from NN is that the algorithms access two index structures (one for each data set) simultaneously. *Mindist* is now defined as the minimum distance between two objects that can lie in the subtrees of two intermediate entries (see Figure 2.2b). If the *mindist* of two intermediate entries E_1 and E_2 (one from each R-tree) is already greater than the distance of the closest pair of objects found so far, the sub-trees of E_1 and E_2 cannot contain a closest pair. Other non-BaB based methods for nearest neighbor search can be found in [KSF+96, SK98, CG99, BEK+98, YOTJ01].

3. TIME-PARAMETERIZED (TP) QUERIES

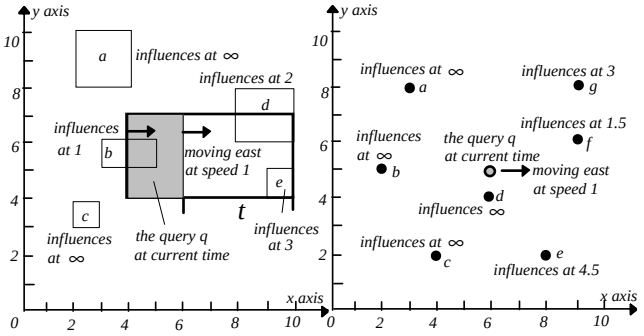
The output of a spatio-temporal TP query has the general form <**R**,**T**,**C**>, where **R** is the set of objects satisfying the corresponding instantaneous query (i.e., current result), **T** is the expiry time of **R**, and **C** the set of objects that will affect **R** at **T**. From the set of objects in the current result **R,** and the set of objects **C** that will cause changes, we can incrementally compute the next result. We refer to **R** as the *conventional,* and (**T**,**C**) as the *time-parameterized* component of the query. Consider, for instance, the TP window query (shaded window) of Figure 3.1a, where objects (rectangle *a* to *e*) are static[1] and query *q* is moving east with speed 1. The output should be <{*b*},1,{*b*}> meaning that object *b* currently intersects the query window, but after 1 time unit it will stop doing so (therefore, *b* should be removed from the result, which will become empty).

A naïve way to process the query is to expand its window so that it includes all the area that the query will cover up to a time *t* in the future, and then process this extended window (using a regular R-tree window query) to find all candidate objects that may change the result up to time *t*. In the example of Figure 3.1a, the extended window (bold rectangle) corresponds to the area that the query will cover in the next *t*=4 time units. For all candidate objects (*b*,*d*,*e*), the interval during which they belong to the result is computed: for *b* this interval is [0,1), for *d* it is [2,4), and for *e* it is [3,4). Given this information we can determine the conventional and the TP components of the query. This method, however, has some serious shortcomings: (i) The estimation *t* of how long in the future to extend the query window is ad-hoc. An under-estimation means that we will not be able to compute the time-parameterized component, while an over-estimation will incur significant computational overhead. (ii) The method is not applicable to other types of queries such as NN.

Observe that the result of a spatial query changes in the future because some objects "influence" its correctness. For instance, if an object (e.g., *b*) satisfies the query at the current time, it may influence the result when it no longer satisfies it in the future (at time 1). On the other hand, an object not currently in the result (e.g., *d*) may influence the query when it becomes a part of the result (at time 2). Figure 3.1a shows the influence time of all

[1] For simplicity of illustration, we often use static 2D objects. The extension to mobile objects and higher dimensions, unless explicitly stated, is straightforward.

objects. Some objects, such as *a* and *c*, may never change the result, in which case the influence time is set to ∞.



(a) TP window query (b) TP NN query

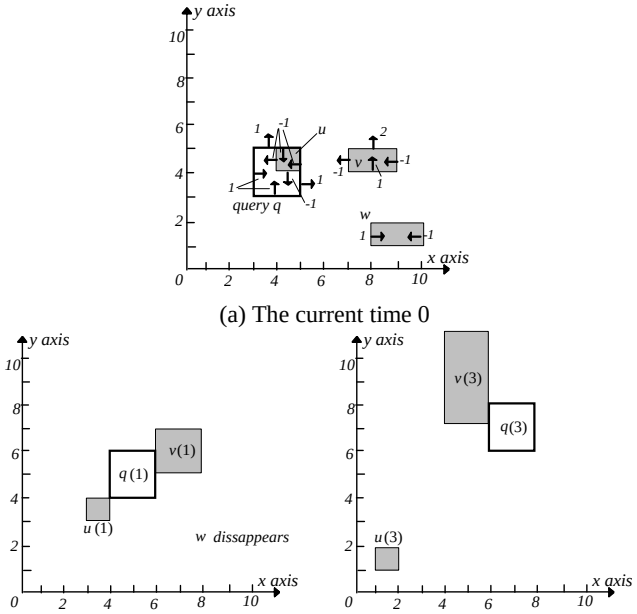
Figure 3.1: The influence time

The concept of "influence time" also applies to other types of queries. Figure 3.1b shows a TP NN, where objects (points *a* to *g*) are static and query point *q* is moving east with speed 1. Point *d* is the current nearest neighbor of *q*. In this case, the influence time of an object should be interpreted as the time that it starts to get closer to the query than the current nearest neighbor. For example, the influence time of point *g* is 3, because at this time *g* will come closer to *q* than *d*. Notice that a non-infinite (i.e., different from ∞) influence time does not necessarily mean that the object will change the result; *g* will influence the query at time 3, only if the result does not change before due to another object (actually at time 3 the nearest neighbor is object *f*). The influence time of points *a*, *b*, *c* is ∞ because they can never be closer to *q* than its current nearest neighbor *d* (observe that the influence time of *d* is also set to ∞).

We denote the influence time of an object *o* with respect to a query *q* as $T_{INF}(o,q)$. The expiry time of the current result is the minimum influence time of all objects. Therefore, the time-parameterized component of a TP query can be reduced to a nearest neighbor problem by treating $T_{INF}(o,q)$ as the distance metric: the goal is to find the objects (**C**) with the minimum T_{INF} (**T**). These are the candidates that may generate the change of the result at the expiry time (by adding to or deleting from the previous answer set). T_{INF} for intermediate entries *E* is defined in a way similar to *mindist* in NN search: $T_{INF}(E,q)$ is the minimum influence time $T_{INF}(o,q)$ of any object *o* that may lie in the subtree of *E*. The above discussion serves as a high-level abstraction that establishes the close connection between the TP retrieval and NN search. In the sequel we derive suitable $T_{INF}(o,q)$ and $T_{INF}(E,q)$ metrics for various query types.

3.1 The TP Window Query

In order to find the influence time $T_{INF}(o,q)$ of an object *o* with respect to a query window *q*, we need the *intersection period* $[T_s,T_e)$ during which *o* will intersect *q*. Figure 3.2a illustrates an example with a dynamic query *q*, and three dynamic objects *u*, *v*, *w* (without loss of generality, assume the current time is 0). Figures 3.2b and *c* show the situations at time 1 and 3 respectively. The intersection period of object *u* is [0,1), of *v* is [1,3), while the intersection period of *w* is [∞,∞). Notice that depending on the values of the two different velocities on a dimension, it is possible that some objects (e.g., *w*) may disappear (i.e., two opposite sides of the rectangle will meet) in the future (time 1). Such objects should be taken into account during query processing, since they may not affect the result after their disappearance.



(a) The current time 0

(b) At time 1 (c) At time 3

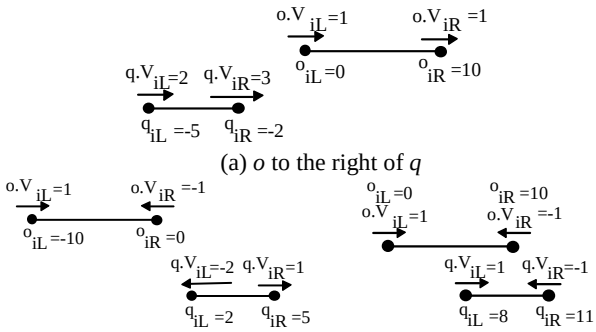
Figure 3.2: Deriving $T_{INF}(o,q)$

We denote the MBR and velocity vector of an object *o* as $\{[o_{1L},o_{1R}],\ldots,[o_{nL},o_{nR}]\}$ and $\{[o.V_{1L},o.V_{1R}],\ldots,[o.V_{nL},o.V_{nR}]\}$ respectively, where $[o_{iL},o_{iR}]$ ($[o.V_{iL},o.V_{iR}]$) corresponds to the extents (velocities) along the *i*th dimension (*i*=1,…,n). The *i*-th projection of an object *o*, will disappear at time $o.T_{iDSP}$ computed as: (i) $o.Ti_{DSP} = \infty$, if $o.V_{iR} \geq o.V_{iL}$ (ii) $o.T_{iDSP} = (o_{iR}-o_{iL})/(o.V_{iL}-o.V_{iR})$, otherwise. The disappearance time $o.T_{DSP}$, is the minimum $o.T_{iDSP}$ of all dimensions. The influence time $T_{INF}(o,q)$ of every object *o* should be no later than $\min(o.T_{DSP},q.T_{DSP})$, after which time either *o* or *q* will have disappeared, thus automatically terminating the intersection period.

Object *o* and query *q* intersect if and only if they intersect along all dimensions. Next we present a method[2] for computing the intersection period $[T_{is},T_{ie})$ along the *i*th dimension, starting with the case where $[o_{iL},o_{iR}]$ does not intersect $[q_{iL},q_{iR}]$ at the current time (i.e., *o* is either totally to the right, or totally to the left of *q*). If *o* is to the right of *q* (Figure 3.3a), then *o* and *q* will start intersecting at the time T_{iLR} (=1) when the leftmost point o_{iL} of *o*, meets the rightmost point q_{iR} of *q*. T_{iLR} is computed as follows: (i) $T_{iLR}=\infty$, if $o.V_{iL} \geq q.V_{iR}$, (i.e., they never meet), and (ii) $T_{iLR} = (o_{iL}-q_{iR})/(q.V_{iR}-o.V_{iL})$, otherwise. Now consider that *o* is to the left of *q* as in Figure 3.3b. In this case, *o* and *q*, will start intersecting at the time T_{iRL} (=2), when the rightmost point o_{iR} of *o*, meets the leftmost point q_{iL} of *q*: (i) $T_{iRL}=\infty$, if $o.V_{iR} \leq q.V_{iL}$, and (ii) $T_{iRL}=(o_{iR}-q_{iL})/(q.V_{iL}-o.V_{iR})$ otherwise. Thus in the general case, the time T_{is} that *o* and *q*, will start intersecting on dimension

[2] TPR-trees also employ a method (narrower in focus and based on different concepts) to compute the intersection period before some designated future time [SJLL00].

i is T_{is} =min(T_{iLR},T_{iRL}), provided of course that o and q do not disappear before (in which case T_{is}=∞) .



(a) o to the right of q

(b) o to the left of q (c) o and q intersect

Figure 3.3: Examples of intersection period

Next we will compute the time T_{ie} that o and q that will stop intersecting on the i-th dimension. In order for [o_{iL},o_{iR}] and [q_{iL},q_{iR}] to stop intersecting, object o must move entirely to the right or to the left of the query. Continuing the example of Figure 3.3a, o and q will keep intersecting from the time (T_{iLR}=1) that o_{iL} meets q_{iR}, till the time (T_{iRL}=15) that o_{iR} meets q_{iL}. On the other hand, in Figure 3.3b, o and q will keep intersecting from the time (T_{iRL}=2) that o_{iR} meets q_{iL}, till the time (T_{iLR}=∞) that o_{iL} meets q_{iR}. Thus, T_{ie} is the maximum of T_{iLR} and T_{iRL}, except for the case that the intersection of period is terminated before due to the object or query disappearance. In general, T_{ie} = min(max(T_{iLR},T_{iRL}), o.T_{DSP}, q.T_{DSP}). In the example of Figure 3.3b, although T_{iLR}=∞, T_{ie}=o.T_{iDSP} =5.

From T_{is} and T_{ie} we can compute the intersection period on all dimensions: [T_s,T_e) = $\bigcap_{i=1,n}[T_{is},T_{ie})$. The influence time $T_{INF}(o,q)$ of an object o not currently intersecting the query, is the earliest time that it will start intersecting, i.e., $T_{INF}(o,q)$=T_s.

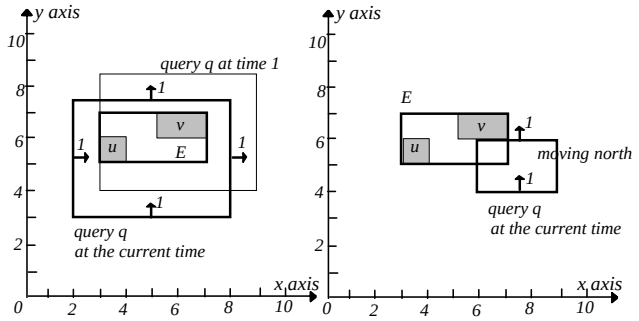
For the case where o and q intersect at the current time, T_{is}=0 for all dimensions, so it remains to derive the end of the intersection period T_{ie}. This is straightforward, based on the observation that o and q will stop intersecting at the first time that either o_{iL} meets q_{iR}, or o_{iR} meets q_{iL}, provided again the query or the object will not disappear before, i.e., T_{ie}=min(T_{iLR}, T_{iRL}, o.T_{DSP}, q.T_{DSP}). In Figure 3.3c, for instance, T_{iLR}=5.5, T_{iRL}=1, o.T_{DSP}=5, q.T_{DSP}=1.5, and T_{ie}=T_{iRL}=1. The end of the intersection period T_e on all dimensions is the minimum T_{ie}, which is also the influence time $T_{INF}(o,q)$ of an object o, currently intersecting the query: $T_{INF}(o,q)$=T_e =min(T_{ie}). Figure 3.4 presents the pseudo-code for computing the intersection period of an object, taking into account disappearance times.

Next we consider $T_{INF}(E,q)$ for an intermediate entry E, which corresponds to the minimum possible influence time of any object in the subtree of E. If the MBR of E does not currently intersect q, $T_{INF}(E,q)$ is the time in the future that E starts to intersect q, because it is also the earliest time when any of the objects inside E can intersect (influence) q. If E intersects q at the current time, we need to distinguish two cases where (i) E is contained in q, or (ii) E partially intersects q. Figure 3.5 illustrates these two cases with static objects u, v, their parent entry E (also static), and a dynamic query q. For the first case (Figure 3.5a), $T_{INF}(E,q)$ is set to the time (=1) that E starts to partially intersect q because, before this time, all objects in E are always contained in q, and hence do not influence the query result (1 is also the influence time of u). For the second case (Figure 3.5b), however, $T_{INF}(E,q)$ must be set to 0 because some object inside E (e.g., v) may influence the result as soon as the query moves.

Compute_Intersection_Period (o, q)
1. [T_s,T_e)=[0,∞]
2. for each dimension i
2. compute disappearance time o.T_{iDSP}, q.T_{iDSP}
3. T_{DSP}=min(o.T_{DSP}, q.T_{DSP})
4. T_{iLR}=(o_{iL}−q_{iR})/(q.V_{iR}−o.V_{iL})
5. if T_{iLR}<0 then T_{iLR}=∞ /*they never meet*/
6. T_{iRL}=(o_{iR}−q_{iL})/(q.V_{iL}−o.V_{iR})
7. if T_{iRL}<0 then T_{iRL}=0 /* they never meet*/
8. if [o_{iL},o_{iR}] does not intersect [q_{iL},q_{iR}]
9. if max(T_{iLR},T_{iRL})≤T_{DSP}
10. T_{is}=min(T_{iLR},T_{iRL}); T_{ie}=max(T_{iLR},T_{iRL})
11. elseif min(T_{iLR},T_{iRL})≤T_{DSP}≤ max(T_{iLR},T_{iRL})
12. T_{is}=min(T_{iLR},T_{iRL}); T_{ie}= T_{DSP}
13. else T_{is}=T_{ie}=∞
14. else /*[o_{iL},o_{iR}] intersects [q_{iL},q_{iR}]*/
15. T_{is}=0
16. T_{ie}=min(T_{iLR},T_{iRL},T_{DSP})
17. [T_s,T_e)=[T_s,T_e)∩[T_{is},T_{ie})
18. return [T_s,T_e)

end Compute_Intersection_Period

Figure 3.4: Algorithm for computing [T_s,T_e)



(a) E is contained in q (b) E partially intersects q

Figure 3.5: Deriving $T_{INF}(E,q)$ when E intersects q

Summarizing, given the intersection period [T_s,T_e) of E and q, which can be computed by the algorithm in Figure 3.4, we define $T_{INF}(E,q)$ as follows:

- $T_{INF}(E,q)$=T_s, if q does not intersect E at the current time (i.e., T_s≠0), or
- $T_{INF}(E,q)$=0, if q intersects, but does not contain, E at the current time, or
- $T_{INF}(E,q)$=$T_{PI}(E,q)$, if q contains E at the current time, where $T_{PI}(E,q)$ is the time that E starts to partially intersect q in the future.

In order to compute $T_{PI}(E,q)$, observe that the containment relation will change to partial intersection at the earliest time T_{iPi} (T_{iPi}∈[T_s,T_e)) such that [E_{iL},E_{iR}] starts to partially intersect [q_{iL},q_{iR}] on any dimension i. This transition will happen at the time T_{iLL} when the leftmost point E_{iL} of E meets the leftmost point q_{iL} of q, or at the time T_{iRR} when E_{iR} meets q_{iR}. The computations of T_{iLL} and T_{iRR} are similar to those of T_{iLR} and T_{iRL} (e.g., T_{iLL}=∞, if E.V_{iL}≥q.V_{iL}, or T_{iLL}=(E_{iL}−q_{iL})/(q.V_{iL}−E.V_{iL}), otherwise). In the example of Figure 3.6, T_{iLL}=2, T_{iRR}=1 and T_{iPi} = T_{iRR}. The partial intersection time T_{iPi} is the minimum of T_{iLL} and T_{iRR}, provided

that o and q do not disappear before. The algorithm for computing T_{PI} is given in Figure 3.7.

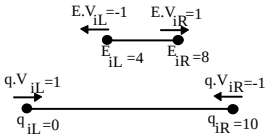


Figure 3.6: Example of partial intersection time

Compute_PI_Time (E, q, [T_s,T_e))
/*call this function if q contains E at the current time; thus T_s=0*/
1. $T_{PI}=\infty$
2. for each dimension i
3. $T_{iLL}=(E_{iL}-q_{iL})/(q.V_{iL}-E.V_{iL})$
4. $T_{iRR}=(E_{iR}-q_{iR})/(q.V_{iR}-E.V_{iR})$
5. if $T_{iLL}\in[T_s,T_e)$ and $T_{iRR}\in[T_s,T_e)$
6. $T_{iPi}=\min(T_{iLL}, T_{iRR})$
7. if $T_{iLL}\in[T_s,T_e)$ and $T_{iRR}\notin[T_s,T_e)$
8. $T_{iPi}= T_{iLL}$
9. if $T_{iLL}\notin[T_s,T_e)$ and $T_{iRR}\in[T_s,T_e)$
10. $T_{iPi}= T_{iRR}$
11. if $T_{iLL}\notin[T_s,T_e)$ and $T_{iRR}\notin[T_s,T_e)$
12. $T_{iPi}= \infty$
13. $T_{PI}=\min(T_{PI},T_{iPi})$
14. return T_{PI}

end Compute_PI_Time

Figure 3.7: Algorithm for computing T_{PI}

Having defined T_{INF} for leaf and intermediate entries, we can employ any BaB algorithm to find the objects o with the minimum influence time $T_{INF}(o,q)$, which is exactly the expiry time of the TP query. Next we address TP KNN queries.

3.2 The TP K Nearest Neighbor Query

We first consider single nearest neighbor (TP NN) queries before extending the solution to an arbitrary number of neighbors. To facilitate understanding, we present our solution for point data in 2D space, although the discussion extends to rectangle objects (where the rationale is the same but the equations more complex). Our analysis focuses on deriving $T_{INF}(o,q)$ and $T_{INF}(E,q)$.

Let P_{NN} be the current nearest neighbor of q. The influence time $T_{INF}(o,q)$ of an object o is the earliest time t in the future such that $o(t)$ starts to get closer to $q(t)$ than $P_{NN}(t)$, where $P_{NN}(t)$, $o(t)$, $q(t)$ are the positions of P_{NN}, o, q at time t respectively. In general, $T_{INF}(o,q)$ is the minimum t that satisfies the following conditions[3]: $\|o(t),q(t)\|\le\|P_{NN}(t),q(t)\|$ and $t\ge0$. If $(o_1,\ldots,o_n)$ are the coordinates, and $(o.V_1,\ldots o.V_n)$ the velocities of a moving point o on dimensions i=1,…,n, the above inequality can be transformed into the standard form $At^2+Bt+C\le0$, where:

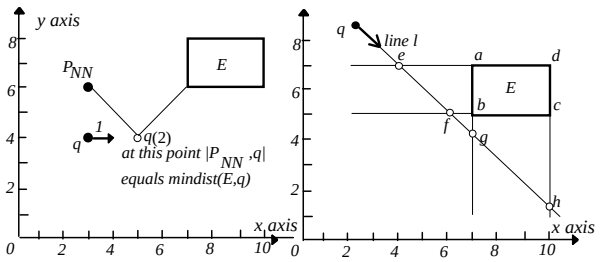
$$A=\sum_{i=1}^{n}\left[\left(o.V_i-q.V_i\right)^2-\left(P_{NN}.V_i-q.V_i\right)^2\right],$$

$$C=\sum_{i=1}^{n}\left[\left(o_i-q_i\right)^2-\left(P_{NN\,i}-q_i\right)^2\right], \text{ and}$$

$$B=\sum_{i=1}^{n}2\left[\left(o_i-q_i\right)\left(o.V_i-q.V_i\right)-\left(P_{NN\,i}-q_i\right)\left(P_{NN}.V_i-q.V_i\right)\right]$$

[3] $\|a,b\|$ denotes the distance between points a and b. Although we use Euclidean distance throughout the paper, other metrics can be applied.

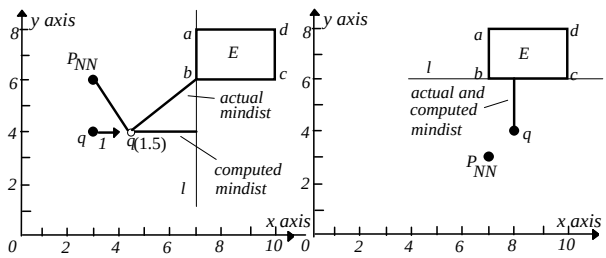
The solution is straightforward and omitted. If no t satisfies the inequality, $T_{INF}(o,q)$ is set to ∞. In case of intermediate entries, $T_{INF}(E,q)$ indicates the earliest time when some object in the subtree of E may start to be closer to q (than P_{NN}). This is illustrated in Figure 3.8a, where P_{NN} and MBR E are static and q is moving east. At time 2, the *mindist* of E to q becomes shorter than $\|P_{NN},q\|$, which implies that some object in E may start to get closer to q (i.e., $T_{INF}(E,q)$=2). More formally, $T_{INF}(E,q)$ is the minimum t that satisfies the condition: $mindist(E(t),q(t)) \le \|P_{NN}(t),q(t)\|$ and $t\ge0$. This inequality requires rather complicated case-by-case discussion because the computation of $mindist(E(t),q(t))$ depends on the relative positions of E and q. Figure 3.8b illustrates an example where the MBR E (corner points a, b, c, d) is static and the query point is moving along line l. Before q reaches point e, $mindist(E,q)$ should be calculated with respect to point a. When q is on the line segment ef, *mindist* is the distance from q to edge ab of E. Similarly, after q passes points f, g, and h, *mindist* should be computed with respect to point b, edge bc, and point c respectively. The situation can be even more complex when, in addition to the query, MBR E is also dynamic, especially in higher dimensional spaces.



(a) Example of $T_{INF}(E,q)$ (b) Different cases of mindist

Figure 3.8: T_{INF} for intermediate entries

Instead, we follow a simpler (but still efficient as shown in the experiments) conservative approach that underestimates *mindist* (to ensure the correctness of BaB algorithms). In particular, we approximate *mindist* with the perpendicular distance from q to a selected edge (or a plane in high-dimensional space) of MBR E. The edge of E is chosen as follows: (i) If the *mindist* at the current time between E and q is with respect to a corner point of E (e.g., point b in Figure 3.9a), then the selected edge (among the two edges connected to the corner point) is the more distant from q (e.g., edge ab is farther to q than bc); (ii) If the *mindist* is computed with respect to an edge (e.g., edge bc in Figure 3.9b), then we select this edge. In this case, the distance from the query point to the edge is exactly the *mindist* at the current time. The pseudocode for the algorithm that applies to arbitrary dimensionality is shown in Figure 3.10; the algorithm returns a (hyper) plane of dimensionality n-1.



(a) Underestimation (b) Accurate estimation

Figure 3.9: Approximate *mindist*

Sel_Plane_Approx_*mindist* (*E*, *q*)

1. if *q* is contained in *E* at the current time
2. return NIL /*no edge selected since $T_{INF}(E,q)=0$*/
3. *sel_dim*=nil /*dimension selected plane is perpendicular to*/
4. *coord*=*velocity*=nil /*the coordinate and velocity of the selected plane on dimension *sel_dim**/
5. *plane_dist*=–∞ /*the distance from *q* to the selected plane at the current time*/
6. for each dimension i
7. if $q_i<E_{iL}$ /*[E_{iL},E_{iR}] is the extent of *E* on dimension *i**/
8. if ($E_{iL}-q_i$>plane_dist) /*q is further to the plane on this dimension than previous dimensions*/
9. *sel_dim*=i; *plane_dist*=$E_{iL}-q_i$
10. *coord*=E_{iL}; *velocity*=$E.V_{iL}$
11. elseif $q_i>E_{iR}$
12. if (q_i-E_{iR}>plane_dist) /*q is further to the plane on this dimension than previous dimensions*/
13. *sel_dim*=i; *plane_dist*=q_i-E_{iR}
14. *coord*=E_{iR}; *velocity*=$E.V_{iR}$
15. return the selected plane (at position *coord* on dimension *sel_dim*, moving at *velocity*)

end Sel_Plane_Approx_*mindist*

Figure 3.10: Selecting a plane to approximate mindist

Without loss of generality, assume that the plane *l* returned by the pseudo-code of Figure 3.10, is perpendicular to the *i*th dimension at point l_i, and moves along the dimension at speed $l.V_i$. $T_{INF}(E,q)$ is the minimum *t* that satisfies the condition $mindist(l(t),q(t)) \leq \|P_{NN}(t),q(t)\|$ and $t\geq 0$. Using the usual notation for *q*, the above inequality is equivalent to:

$$|(q_i-l_i)+t\cdot(q.V_i-l.V_i)| \leq \sqrt{\sum_{i=1}^{n}\left[(P_{NNi}-q_i)+t\cdot(P_{NN}.V_i-q.V_i)\right]^2}$$

which can be transformed to the standard (and easily solvable) form $At^2+Bt+C\leq 0$, where

$$A=(q.V_i-l.V_i)^2-\sum_{i=1}^{n}(P_{NN}.V_i-q.V_i)^2 \quad C=(q_i-l_i)^2-\sum_{i=1}^{n}(P_{NNi}-q_i)^2$$

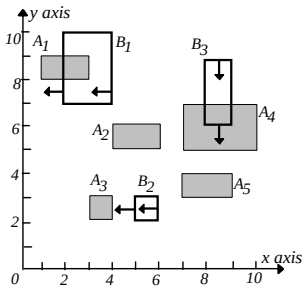
and

$$B=2(o_i-q_i)(o.V_i-q.V_i)-\sum_{i=1}^{n}2(P_{NNi}-q_i)(P_{NN}.V_i-q.V_i)$$

The extension to TP KNN queries is straightforward. The only difference is that now the influence time of an object *o* corresponds to the earliest time that *o* starts to get closer to *q* than any of the K current neighbors. Specifically, assuming that the K current neighbors are P_{NN1}, P_{NN2},…, P_{NNK}, we first compute the influence time T_{INFi} of *o* with respect to each P_{NNj} (j=1,2,…,K) following the previous approach. Then, $T_{INF}(o,q)$ is set to the minimum of T_{INF1}, T_{INF2}, …, T_{INFK}. Similarly, for $T_{INF}(E,q)$ we first compute the T_{INFj} of *E* with respect to each P_{NNj} and then set $T_{INF}(E,q)$ to the minimum of T_{INF1}, T_{INF2}, …, T_{INFK}.

3.3 The TP Join Query

A join query returns all pairs of objects from two datasets that satisfy some spatial condition (e.g., intersection). The join result changes in the future when: (i) a pair of objects in the current result, ceases to satisfy the join condition, or (ii) a pair not in the result starts to satisfy the condition. Figure 3.11a shows an example of TP join. Objects A_3 and B_2, which do not intersect at the current time, will start intersecting at time 1, hence influencing the result. In general, we denote the influence time of a pair of objects (o_1,o_2) as $T_{INF}(o_1,o_2)$. Figure 3.11b lists the T_{INF} for all pairs of objects. The influence time is ∞, if a pair will never change the join result (e.g., (A_2,B_2)). The expiry time is the minimum influence time (i.e., $T_{INF}(A_3,B_2)=1$). As in the other types of TP queries, by adding or deleting the pair(s) of objects (A_3,B_2) that cause(s) the change, the join result is updated incrementally.



(a) A TP window join example

	A_1	A_2	A_3	A_4	A_5
B_1	3	∞	∞	∞	∞
B_2	∞	∞	1	∞	∞
B_3	∞	∞	∞	4	2

(b) Influence time of all pairs

Figure 3.11: Influence time of object pairs

A TP join can be regarded as a closest pair (CP) query (see section 2.2) by treating $T_{INF}(o_1,o_2)$ as the distance metric between objects o_1 and o_2. In addition, we need to define $T_{INF}(E_1,E_2)$ to replace $mindist(E_1,E_2)$ (see Figure 2.2b), where $T_{INF}(E_1,E_2)$ is the minimum $T_{INF}(o_1,o_2)$ of all pairs formed by any two objects o_1 and o_2 in the subtrees of E_1 and E_2 respectively. The analysis is very similar to that for TP window queries. We simply summarize the definitions:

- $T_{INF}(o_1,o_2)=T_e$, if $T_s=0$ (i.e., o_1 and o_2 currently satisfy the join condition), or $T_{INF}(o_1,o_2)=T_s$, if $T_s>0$ (i.e., o_1 and o_2 do not satisfy the condition), where $[T_s,T_e)$ is the intersection period of objects o_1 and o_2
- $T_{INF}(E_1,E_2)=T_s$, where T_s is the starting point of the intersection period $[T_s,T_e)$ of E_1 and E_2 (unlike TP window queries, this case also includes containment)

The intersection period $[T_s,T_e)$ for object and intermediate entry pairs is computed by the algorithm of Figure 3.4.

3.4 Query Processing

Both depth- and best-first search (as discussed in section 2) can be used for processing TP queries. Figure 3.12 (DF) and 3.13 (BF) show the pseudo-code for window queries. The algorithms use three global variables **R**, **T** and **C** to store the three outcomes of a query. In order to obtain the current result (**R**), both algorithms visit entries that intersect the original window although the T_{INF} of these entries maybe greater than the minimum influence time (**T**). Furthermore, we need to distinguish between (i) $T_{INF}(o,q)<$**T** and (ii) $T_{INF}(o,q)=$**T**. In the first case, *o* becomes the only object that influences the result so far, while in the second case *o* is added to the set of influencing objects **C** (it is possible that multiple objects will enter or exit the query window at the same time). The algorithms for TP joins are similar to those of CP queries. In particular, they traverse the R- (or TPR-) trees of the two datasets simultaneously, following pairs of intermediate entries (E_1,E_2), if one of the following conditions holds: (i) the MBRs of E_1 and E_2 intersect (so some objects may satisfy the join condition in their subtrees), or (ii) $T_{INF}(E_1,E_2)$ is less than the minimum influence time of all object pairs seen so far (in this case their subtrees may contain object pairs that trigger the next result change).

```
Depth-first_TP_Window_Query (current node N)
/*invoke by passing the root of R-tree*/
/*initially: T=∞, R=Ø, C= Ø */
1.    if N is a leaf
2.    for each object o
3.      if T_INF(o,q)<T
4.        C={o}
5.        T= T_INF(o,q)
6.      else if T_INF(o,q)=T
7.              C=C∪{o}
8.      if o intersects q then R=R ∪{o}
      /*o satisfies the original window query*/
9.  else /*N is an intermediate entry*/
10.   sort all the entries E by their T_INF(E,q)
11.   for each entry E
12.     if (T_INF(E,q)≤T) or (E.MBR intersects q)
13.       TP_Window_Query (e.childnode)
end Depth-first_TP_Window_Query
```

Figure 3.12: DF algorithm for TP window queries

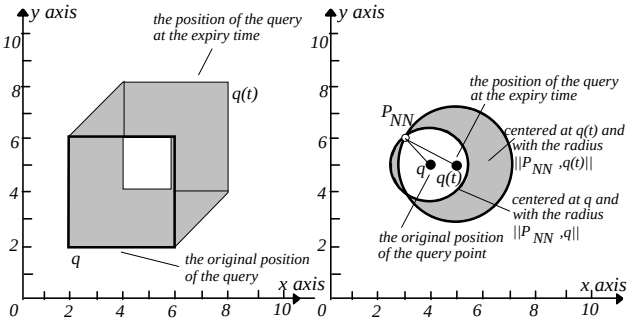
```
Best-first_TP_Window_Query ( )
1.  initialize a heap H that accepts <key,entry>
2.  retrieve the root node R
3.  for each entry E in R insert <T_INF(E,q),E> to H
4.  while (H is not empty)
5.    de-heap<key,E>/*E has the minimum key in H*/
6.    if E points to a leaf node
7.      for each object o in E.childnode
8.        if T_INF(o,q)<T
9.          C={o}
10.         T= T_INF(o,q)
11.       else if T_INF(o,q)=T
12.             C= C∪{o}
13.       if o intersects q then R=R ∪{o}
14.   else /*E points to an intermediate node*/
15.     if T_INF(E,q) ≤T
16.       for each entry E' in E.childnode
17.         insert <T_INF(E',q),E'> to H
18.     else  /* T_INF(E,q) >T */
19.       search E.subtree for objects that intersect q
end Best-first_TP_Window_Query
```

Figure 3.13: BF algorithm for TP window queries

Although for TP windows and joins the conventional and the time-parameterized components of a query can be obtained in one pass, TP KNN processing requires the retrieval of **R** before **T** and **C**, since the validity period and the objects that influence the result depend on the current nearest neighbors. Thus, while TP window queries should have about the same cost as their traditional counterparts, TP KNN are expected to be more expensive.

Notice that in dynamic databases, object updates may change the time-parameterized component of the query result. In this case, instead of re-evaluating the query for each update, we can compute the new influence time of the updated object *o*. If the new $T_{INF}(o)$ is before the expiry time of the current query result, we set the validity time to $T_{INF}(o)$ and object *o* as the next result change. On the other hand, if $T_{INF}(o)$ is later than the expiry time, we ignore this update.

Finally, the reduction framework facilitates performance analysis by utilizing previous findings on nearest neighbor search. Recall from section 2.2, that an optimal NN algorithm only needs to visit those nodes, whose MBRs intersect the "search region" around the query point. Such search regions also apply for processing the time-parameterized component (retrieval of **T** and **C**) of TP queries. Assuming that O_{NN} is the object with the minimum influence time, all entries *E* to be visited by an optimal algorithm should satisfy the condition: $T_{INF}(E,q) \le T_{INF}(O_{NN},q)$. Based on this, Figures 3.14a and b demonstrate the corresponding search regions (shaded areas) of TP window and NN queries for static datasets (for dynamic datasets, the search region changes with time). The white areas do not belong to the search regions.



(a) TP window query (b) TP NN query

Figure 3.14: Search regions for static data sets

4. EXTENSIONS

The general concept of TP queries and the associated processing mechanisms are directly relevant to several other types of queries. Section 4.1 discusses continuous spatio-temporal queries with arbitrary terminating clauses, while section 4.2 focuses on earliest event queries that find the first future time that a specific event may happen (e.g., "towards which direction should I move to catch the first moving bus?").

4.1 Continuous Spatio-temporal Queries

A continuous query returns a set of <**R**,**T**> tuples, where each **R** corresponds to a result satisfying the query, and **T** its validity period. The termination clause determines (explicitly or implicitly) a future time up to which the query should be evaluated. An example is shown in Figure 4.1, where the goal is to "find my NNs (i.e., gas stations) during my trip from *s* to *e*, through intermediate point *p*". The result should be $<\{a\}, [s, p_1)>$, $<\{b\}, [p_1, p_2)>$, $<\{c\}, [p_2, e)>$, meaning that *a* will be my NN during $[s, p_1)$, *b* during $[p_1, p_2)$ and so on. The only existing way to process this query, is by executing numerous incremental NN queries at pre-defined sample points [SR01]. The shortcomings of this approach are discussed in section 2.

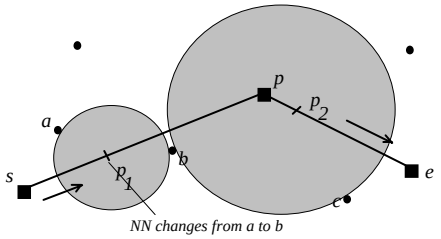


Figure 4.1 Example of continuous NN query

On the other hand, our framework provides a more natural and efficient method: execute a time parameterized NN query at point

s, to get the first NN (**R**={*a*}), the validity period of the result (**T** corresponds to point p_1) and the next nearest neighbor (**C**={*b*}). Then, retrieve the TP component (i.e., **C** and **T**) at the points where there is a change in the result (i.e., p_1 and p_2), or the points where is a change in the query direction (i.e., *p*). That is, the processing of the query involves one (regular) NN search, and four (including the point of origin) computations of the TP component. This *repetitive* approach can be applied for any query type and terminating conditions, such as "stop when the result changes *n* times", "stop when the result contains *n* objects", "stop when the query reaches a certain point in space", etc.

More efficient methods are possible for continuous queries where the influence time of an object does not depend on the other objects, but remains constant throughout the lifespan of the query (e.g., TP windows, joins). Consider for instance, a variation of the previous example where the goal is to "find the points within 1km range during my route from *s* to *e*, through intermediate point *p*" (see Figure 4.2). The result (<Ø, [*s*, p_1)>, <{*a*}, [p_1, p_2)>, < Ø, [p_2, p_3)>, <{*b*}, [p_3, p_4)>, …) can be determined by applying twice a slightly modified version of the BF *TP_Window* algorithm. The first application will retrieve all objects intersecting W_1 (all the area that the query range will cover from *s* to *p*), while the second application will cover W_2 (the area covered from *p* to *e*). When an object is encountered, two influence times (the beginning and end of its satisfaction period) are inserted into the heap until the termination condition holds. The result is then easily obtained by the order of the objects in the heap. This *continual* approach can also be modified for various terminating conditions, but is not applicable to queries (e.g., NN) where the influence times change during the query.

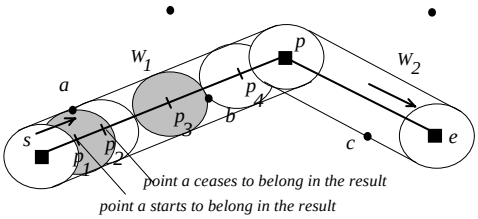
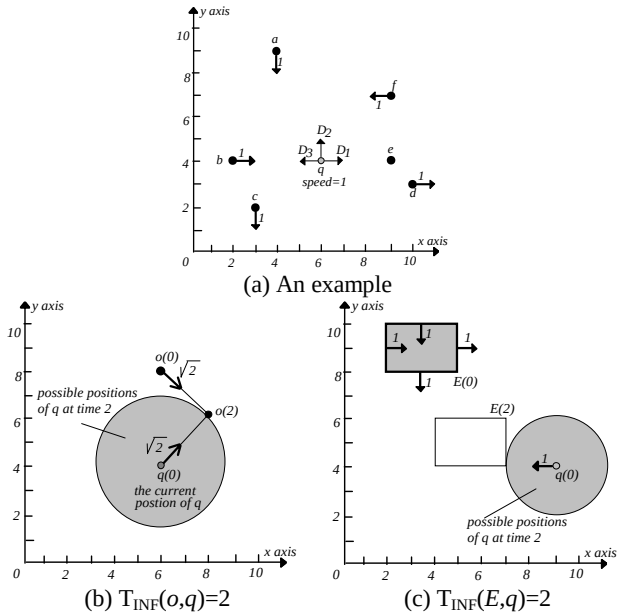


Figure 4.2 Example of continuous window query

4.2 Earliest Event Queries

An e*arliest event* query retrieves the first future time that a certain "event" can happen in some dynamic environment. Although such queries can not be characterized as continuous (they return a single result with no validity period), their processing is directly related to TP queries. Figure 4.3a shows an example, where the goal is to decide a movement direction for the query point *q* (whose maximum speed is 1) such that *q* can "catch" one of the points as soon as possible (e.g., a person trying to catch a bus). In this example, if *q* moves towards D_1, D_2, and D_3, the first points that it will encounter are *e*, *f*, and *b* (at time 3, 3, 2) respectively. It can be easily verified that direction D_3 is indeed the direction towards which *q* can catch the earliest point.

Earliest event queries can also be reduced to nearest neighbor search by defining appropriate T_{INF}. At any future time *t*, all the possible positions that can be reached by the query point *q* constitute a *vicinity circle* centered at *q*(0) (i.e., the initial position of query q) with radius $t \cdot q.V$ (*q*.V is the maximum velocity of *q*). Figure 4.3b demonstrates the vicinity circle of *q* at time 2. The earliest time $T_{INF}(o,q)$ that an object *o* can be caught by *q* (i.e., *o* falls into the vicinity circle of *q*) is the minimum *t* for which: $\|o(t),q(0)\| \le t \cdot q.V$ and $t \ge 0$. For intermediate entries, $T_{INF}(E,q)$ corresponds to the earliest time that *q* can catch any point covered by the MBR of *E*, which is the earliest time *t* such that *E* intersects the vicinity circle of *q* at *t* (Figure 4.3c shows a case where $T_{INF}(E,q)=2$). Thus, $T_{INF}(E,q)$ is the minimum *t* that satisfies the conditions $mindist(E,q) \le t \cdot q.V$ and $t \ge 0$. Both these inequalities can be solved as shown in section 3.



(a) An example

(b) $T_{INF}(o,q)=2$ (c) $T_{INF}(E,q)=2$

Figure 4.3: An earliest event problem

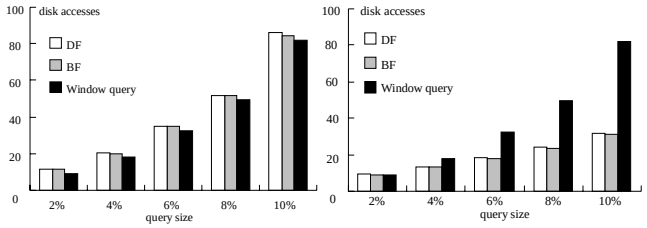
5. EXPERIMENTAL EVALUATION

In this section, we evaluate the efficiency of the proposed methods through extensive experimentation with static and dynamic datasets (queries are always dynamic). As static datasets we use the LA file [Tiger], which contains 130K MBRs, and the CA file [Sequoia] that contains 64K points. Due to limited availability of real datasets of moving objects, we generated dynamic datasets (density=0.5) where all velocity components of objects distribute uniformly in [–0.1, 0.1]. The cardinality ranges between 10K and 100K, while the distribution of objects at the current time can be Gaussian or uniform in a unit universe. In the sequel, we refer to a synthetic rectangle dataset as $RD_{DIST,\ CARD}$, where DIST and CARD are its distribution and cardinality. Similarly, synthetic point datasets are denoted as $PD_{DIST,\ CARD}$.

The R- and TPR-tree implementations are based on [BKSS90] and [SJLL00], respectively. The disk page is set to 1K bytes. With this size, the node capacity in R- (TPR-) trees is 48 (26). Unless stated otherwise, an LRU buffer with 50 pages is assumed. Performance is measured by the average number of disk accesses in performing workloads of 200 dynamic queries. The positions of queries in a workload conform to the distribution of the queried dataset in order to avoid queries in empty space. The query velocities range uniformly in [–0.1, 0.1]. All window queries in a workload have the same side length, denoted as a percentage of the universe extent. We first present the results for R-trees on static datasets, followed by TPR-trees on dynamic datasets.

5.1 Static Datasets

The first set of experiments evaluates the performance of TP window queries using LA dataset. Since a TP query retrieves more information than its conventional component, it is at least as expensive. In order to assess the additional cost, we perform TP window queries with extents ranging from 2% to 10% (i.e., covering up to 1% of the spatial universe). Figure 5.1a compares the number of page accesses using the BF and DF approach, with that of the regular queries. Both DF and BF incur marginal overhead (2-3 I/Os). This is expected because in case of TP windows (and joins) the conventional and TP components are processed in a single pass. The objects with the minimum influence time, are most often inside nodes that intersect the query window now, and therefore will be retrieved by the conventional component anyway.



(a) Cost of TP window queries (b) Cost of TP component

Figure 5.1: TP window query evaluation for static datasets

In order to further analyze TP window queries, we evaluate the same workloads with only the time-parameterized components, i.e., we retrieve the expiry time and change, but not the current result. As shown in Figure 5.1b, processing the TP component is usually cheaper than regular spatial queries, and the difference increases with the query window. This is explained by the fact that the search area of a TP window query (Figure 3.14a) is usually very small, especially when the object triggering the change is close.

The exclusive retrieval of the TP component may be performed by the *repetitive* approach (as discussed in section 4.1) during the evaluation of complex continuous queries. In case of window queries however, the *continual* approach (also discussed in section 4.1) is obviously more efficient since it only performs one query (provided that the velocity vector of the query remains constant). Figure 5.2 demonstrates the page accesses of the continual approach for window queries (extent 6%) as a function of the number of result changes retrieved.

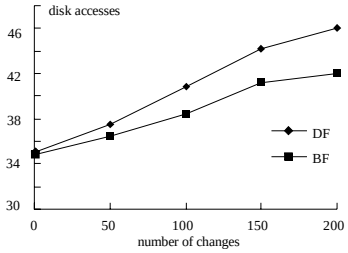
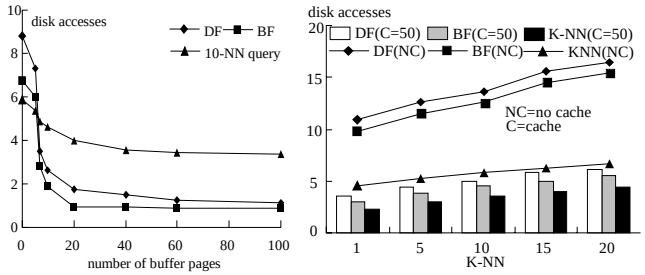


Figure 5.2: Continual window queries for static datasets

We now proceed to evaluate TP KNN queries, using point dataset CA. Recall that processing a TP KNN query is always divided into an ordinary KNN query (the first pass), followed by the TP component (the second pass). Figure 5.3a compares the performance of the two passes (for TP 10-NN queries) as a function of number of (LRU) buffer pages. When there is no buffer, the second pass requires more disk accesses; however, the performance of the second step improves fast even with a very small buffer. This is because the two passes have similar access patterns, and pages loaded for the conventional component are later available for TP processing. This is further confirmed in Figure 5.3b, which shows the cost of a complete TP query versus that of an ordinary KNN query (costs are shown as a function of K). Notice that, when there is no buffer, a TP query is significantly more expensive than the corresponding KNN query. The addition of a buffer with C=50 pages, reduces this difference considerably; the cost of a BF TP KNN is only 10%-20% higher than that of the regular query.



(a) Cost of TP component (b) Cost of TP KNN queries

Figure 5.3: TP KNN query evaluation for static datasets

Figure 5.4 evaluates the performance of continuous (single) NN queries (similar to the example of Figure 5.1a) as a function of the result changes (e.g., how many times the NN neighbor will be updated during the life span of the query). The cost of the repetitive approach grows linearly with the number of query changes retrieved. In comparison to Figure 5.2, the growth is faster because now each change triggers a new TP query, while in the continual approach the number of changes only affects performance implicitly by increasing the extents of the query window in the future. Nevertheless, as discussed before, the continual approach is not applicable for TP NN.

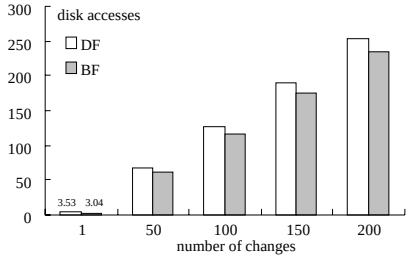


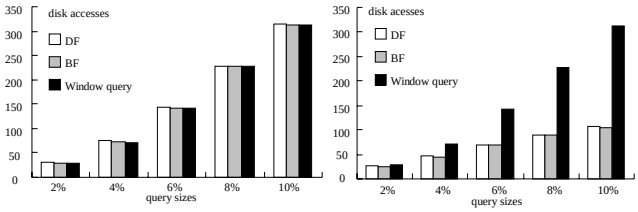
Figure 5.4: Performance of continuous KNN queries

TP joins are meaningless for static datasets, since at least one dataset must be dynamic in order to change the result. Dynamic datasets are evaluated in the next section.

5.2 Dynamic Datasets

In order to test the validity and generality of our observations, we repeated the experiments of the previous section using dynamic datasets $DS_{GAU,100K}$ and $PS_{GAU,100K}$ (with 100K rectangles and points respectively) indexed by TPR-trees. Figures 5.5 to 5.8 correspond to the diagrams in Figures 5.1 to 5.4. The results are very similar (with dynamic datasets being, in general, more expensive to process) and we simply outline the conclusions: (i) TP windows involve almost the same cost as their traditional

counterparts, since they, more or less, access the same nodes, (ii) TP KNN are more expensive than regular KNN queries, but the cost difference is insignificant if a (small) buffer is used, (iii) BF outperforms DF, but the gain is important only for continual queries that extend far into the future (iv) the continual approach, whenever applicable, is preferable to the repetitive method of processing continuous queries.



(a) Cost of TP window queries (b) Cost of TP component

Figure 5.5: TP window query evaluation for dynamic datasets

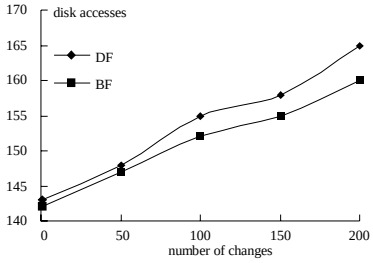
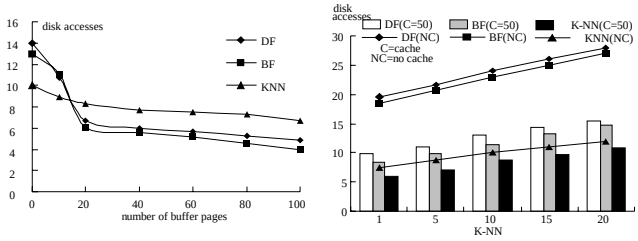


Figure 5.6: Continual window queries for dynamic datasets



(a) Cost of TP component (b) Cost of TP KNN queries

Figure 5.7: TP KNN query evaluation for dynamic datasets

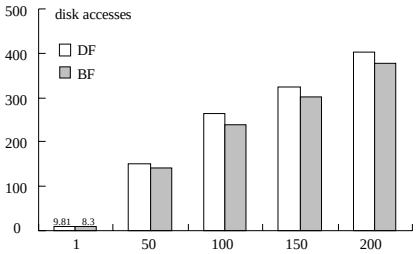
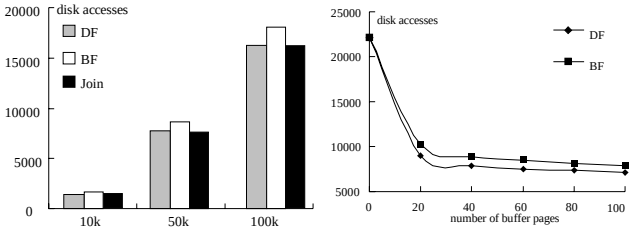


Figure 5.8: Performance of continuous KNN queries

For testing TP Joins, we generated several dynamic rectangle datasets with different distributions and cardinalities: $DS_{GAU,10K}$, $DS_{GAU,50K}$, $DS_{UNI,10K}$, $DS_{UNI,50K}$, $DS_{UNI,100K}$. The first experiment (Figure 5.9a) performs TP joins on uniform and Gaussian datasets of the same cardinality, and compares the costs (page accesses) with that of ordinary spatial joins (implemented based on [BKS93]) as a function of cardinality. TP joins are slightly more expensive because: (i) some extra nodes should be accessed for objects producing the next result change, and (ii) TP joins deploy different visiting orders from ordinary joins, which involve several heuristics to improve the access locality and utilize the buffer [BKS93].

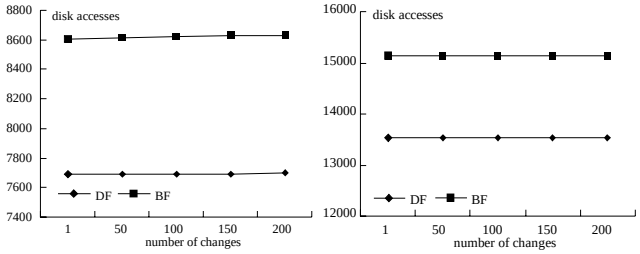
An interesting observation is that, unlike TP window and KNN queries, for TP joins BF is outperformed by DF. This is due to the fact that best-first traversal leads to worse access locality; thus, it is favored less by buffers (recall that we use a buffer of 50 pages). The same phenomenon was observed in [CMTV00] for closest pair queries. Figure 5.9b further confirms this by illustrating the costs of the two approaches under various buffer sizes (for joining $DS_{GAU,50K}$, $DS_{UNI,50K}$). For zero buffer, DF and BF have almost the same cost, but the addition of a small buffer is more beneficial to DF.



(a) TP join cost vs. cardinality (b) TP join cost vs. buffer size

Figure 5.9: TP join evaluation for dynamic datasets

Next we study continuous joins, where we retrieve the current result and the subsequent 1,..,200 changes. Figure 5.10a shows the number of page accesses as a function of the number of result changes. The cost is almost constant, even if up to 200 changes are retrieved. This is explained by the fact that the most important factor in the total cost is the conventional component (i.e., retrieval of the current result). The TP component is minimal in comparison and does not affect the result significantly. Finally, we join a dynamic dataset ($DS_{GAU,100K}$) with a static one (LA). The results are similar to that of Figure 5.10a and in both cases DF outperforms BF.



(a) Dynamic datasets (b) Dynamic/static dataset

Figure 5.10: Continuous TP joins

6. CONCLUSION

Regular spatial queries are of limited use in dynamic environments, unless the results are accompanied by an expected validity period. In this paper we propose a general framework for transforming any spatial query to a time-parameterized version which, in addition to the current result, returns its expiry time and the changes. As shown in the experimental evaluation, the extra information is obtained at zero or minimal cost. We believe that our techniques are crucial for many emerging applications that deal with spatio-temporal data, such as mobile communications and weather prediction. The contributions of the paper are summarized as follows:

- Introduction of the novel concept of time-parameterized queries.
- Techniques for transforming the most common spatial queries to their TP counterparts.
- Development of efficient processing methods.
- Application to other query types such as continuous and earliest event queries.

Although we tried to cover several issues, there still exist numerous challenging problems and directions for future work. An obvious one is the extension to other query types. For example, a TP closest pair (TP CP) query identifies future changes in the closest pairs of objects from two dynamic datasets (e.g., "inform a set of customers about when their nearest cabs will change"). As with TP KNN queries, the influence time of a pair of objects (customer, cab) in the TP CP problem depends on the closest pair now. Thus, an efficient definition for $T_{MIN}(E_1,E_2)$ is difficult, because it requires the knowledge of nearest cabs of all customers.

Furthermore, notice that several queries discussed in this paper can be formulated as computational geometry problems. Continuous KNN, for example, can be defined as follows: given a set of points and a query trajectory, retrieve all points that are among the K nearest neighbors of any point on the trajectory. Our solution (based on the repetitive approach) is output-sensitive. There may exist other methods (e.g., extensions of Voronoi diagrams?) where the result is independent of the number of changes and, therefore, they may be preferable for long trajectories. In any case, it would be interesting to obtain theoretical bounds for the performance of TP and continuous spatio-temporal queries.

ACKNOWLEDGEMENTS

This work was supported by grants HKUST 6081/01E and HKUST 6070/00E from Hong Kong RGC.

REFERENCES

[AAE00] Agarwal, P.K., Arge, L., Erickson, J. Indexing Moving Points. *ACM SIGMOD*, 2000.

[BBKK97] Berchtold, S., Bohm, C., Keim, D.A., Kriegel, H. A Cost Model for Nearest Neighbor Search in High-Dimensional Data Space. *ACM PODS*, 1997.

[BEK+98] Berchtold, S., Ertl, B., Keim, D., Kriegel, H., Seidl, T. Fast Nearest Neighbor Search in High-Dimensional Space. *IEEE ICDE*, 1998.

[BBK+01] Berchtold, S., Bohm, C., Keim, D., Krebs, F., Kriegel, H.P. On Optimizing Nearest Neighbor Queries in High-Dimensional Data Spaces. *ICDT*, 2001.

[BJSS98] Bliujute, R., Jensen, C.S., Saltenis, S., Slivinskas, G. R-tree Based Indexing of Now-Relative Bitemporal Data. *VLDB* 1998.

[BKS93] Brinkhoff, T., Kriegel, H.P., Seeger, B. Efficient Processing of Spatial Joins Using R-trees. *ACM SIGMOD*, 1993.

[BKSS90] Beckmann, N., Kriegel, H.P., Schneider, R., Seeger, B. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. *ACM SIGMOD*, 1990.

[CDTW00] Chen, J., DeWitt, D.J., Tian, F., Wang, Y. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. *ACM SIGMOD*, 2000.

[CG99] Chaudhuri, S., Gravona, L. Evaluating Top-K Selection Queries. *VLDB*, 1999.

[CMTV00] Corral, A., Manolopoulos, Y., Theodoridis, Y., Vassilakopoulos, M. Closest Pair Queries in Spatial Databases. *ACM SIGMOD*, 2000.

[CPZ98] Ciaccia, P., Patella, M., Zezula, P. A Cost Model for Similarity Queries in Metric Spaces. *ACM PODS*, 1998.

[HS99] Samet, H., Hjaltason, G. Distance Browsing in Spatial Databases. *ACM TODS*, 1999.

[KGT99] Kollios, G., Gunopulos, D., Tsotras, V. On Indexing Mobile Objects. *ACM PODS*, 1999.

[KSF+96] Korn, F., Sidiropoulos, N., Faloutsos, C., Siegel, E, Protopapas, Z. Fast Nearest Neighbor Search in Medical Image Databases. *VLDB*, 1996.

[PM97] Papadopoulos, A., Manolopoulos, Y. Performance of Nearest Neighbor Queries in R-trees. *ICDT*, 1997.

[RKV95] Roussopoulos, N., Kelly, S., Vincent, F. Nearest Neighbor Queries. *ACM SIGMOD*, 1995.

[Sequoia] http://dias.cti.gr/~ytheod/research/datasets/spatial.html.

[SJLL00] Saltenis, S., Jensen, C., Leutenegger, S., Lopez, M. Indexing the Positions of Continuously Moving Objects. *ACM SIGMOD*, 2000.

[SK98] Seidl, T., Kriegel, H. Optimal Multi-Step K-Nearest Neighbor Search. *ACM SIGMOD*, 1998.

[SR01] Song, Z., Roussopoulos, N. K-Nearest Neighbor Search for Moving Query Point. *SSTD*, 2001.

[SWCD97] Sistla, P., Wolfson, O., Chamberlain, S., Dao, S. Modeling and Querying Moving Objects. *IEEE ICDE*, 1997.

[TGNO92] Terry, D., Goldberg, D., Nichols, D., Oki, B. Continuous Queries over Append-Only Databases. *ACM SIGMOD*, 1992.

[Tiger] http://dias.cti.gr/~ytheod/research/datasets/spatial.html.

[TUW98] Tayeb, J., Ulusory, O., Wolfson, O. A Quadtree Based Dynamic Attribute Indexing Method. *The Computer Journal*, Vol. 41(3), pp., 185-200, 1998.

[WSB98] Weber, R., Schek, H., Blott, S. A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces. *VLDB*, 1998.

[YOTJ01] Yu, C., Ooi, B.C., Tan, K.L., Jagadish, H.V. Indexing the Distance: An Efficient Method to KNN Processing. *VLDB*, 2001.

[ZL01] Zheng, B., Lee, D. Semantic Caching in Location-Dependent Query Processing. *SSTD*, 2001.